# CMSC201
# Computer Science I for Majors

## Lecture 10 – Functions (cont)

All materials copyright UMBC and Dr. Katherine Gibson unless otherwise noted

# Last Class We Covered

- Functions
  - Why they're useful
  - When you should use them
- Calling functions
- Variable scope
- Passing parameters

# Any Questions from Last Time?

# Quick Announcement

- Update made to the output (but not the directions) for Homework 4, Part 2
- Exclamation marks are valid if they appear anywhere in the password, not only at the end

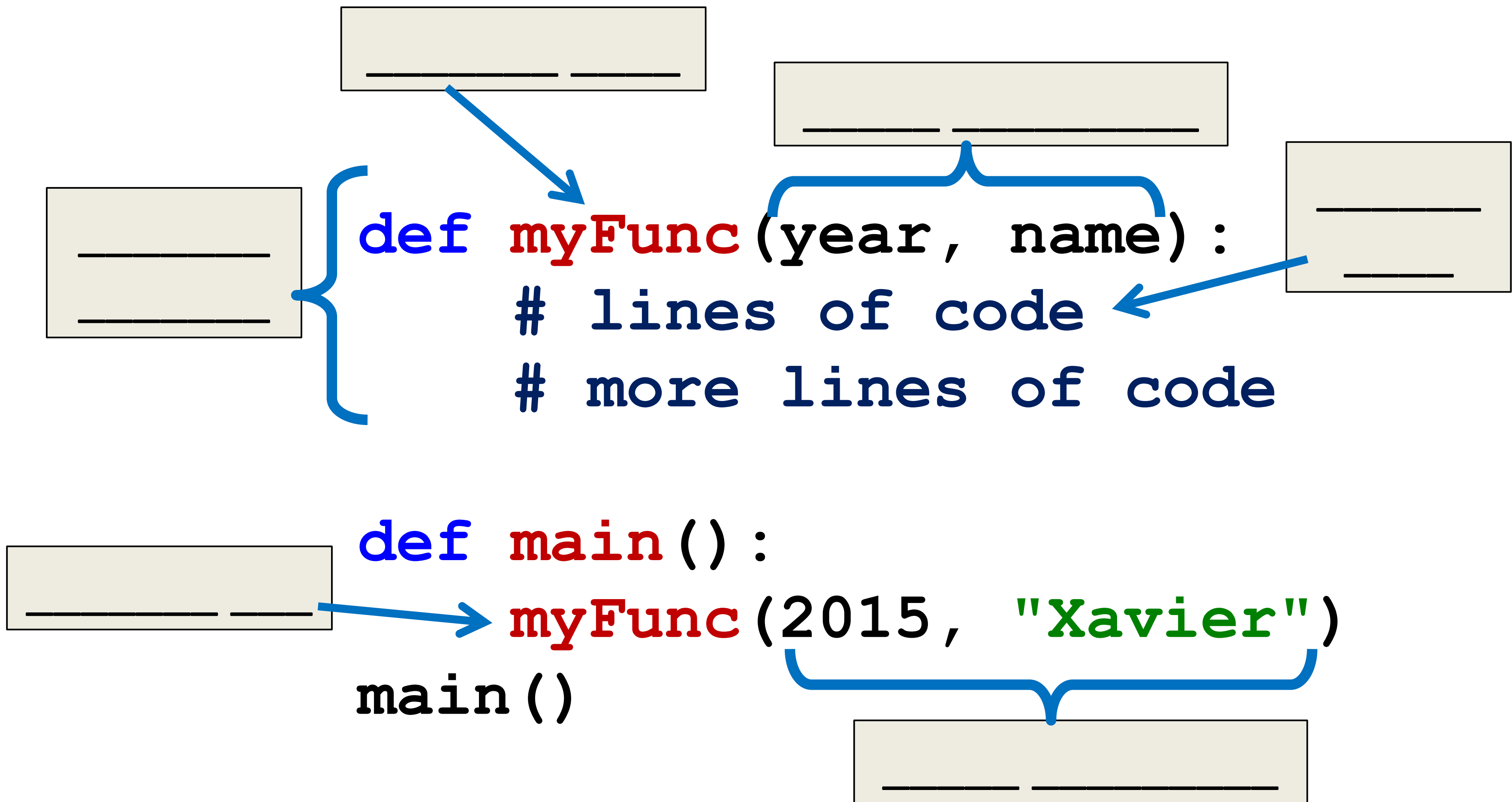
```
Please enter a password: sciencerules
The password is all lowercase, so it must contain
the character ! to be secure.
Please enter a password: science!rules
Thanks for picking the password science!rules
```

# Today's Objectives

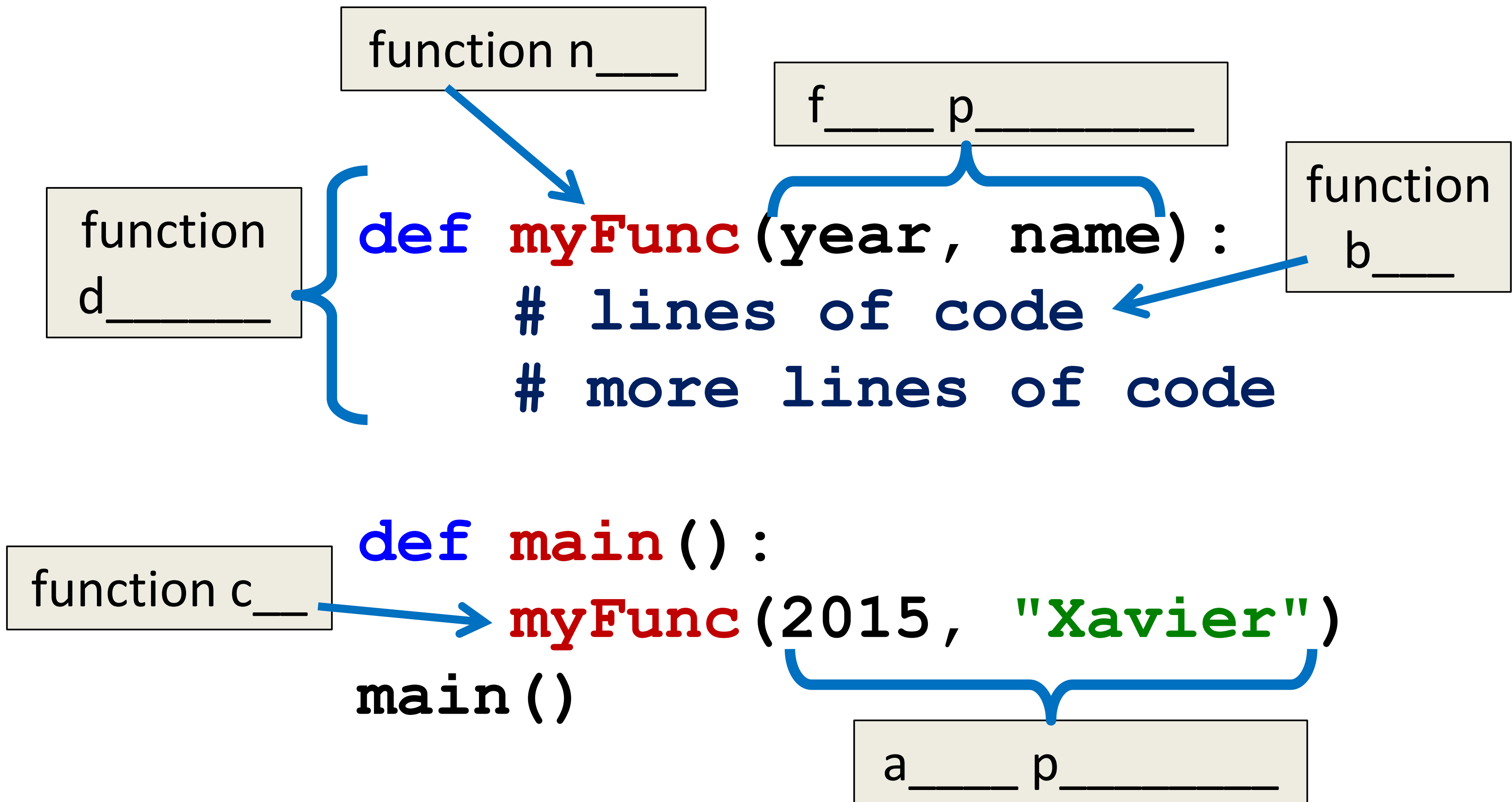
- To introduce value-returning functions
- To better grasp how values in the scope of a function actually work
- To understand mutability (and immutability)
- To practice function calls

# Review: Parts of a Function

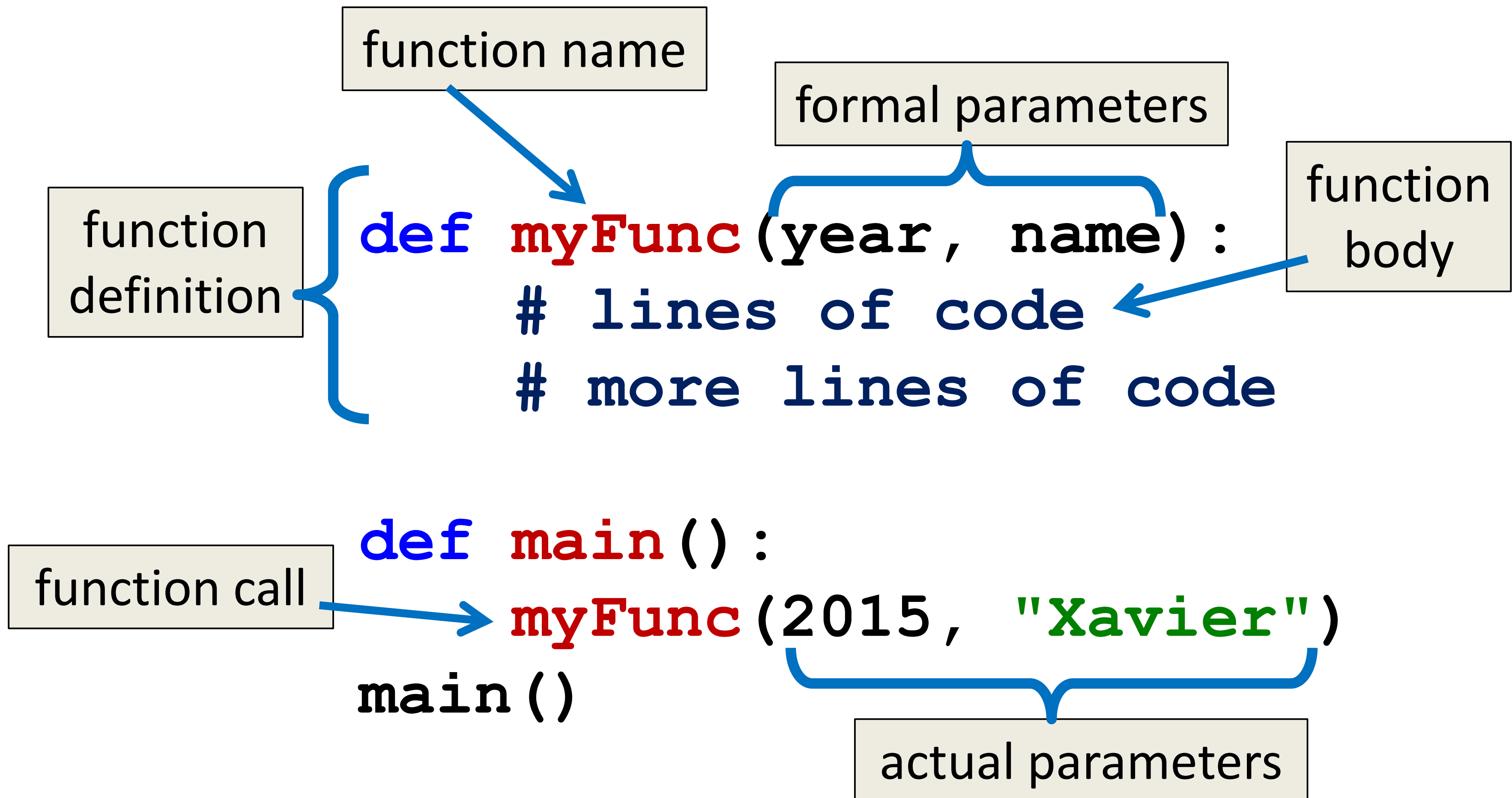
# Function Vocabulary



# Function Vocabulary

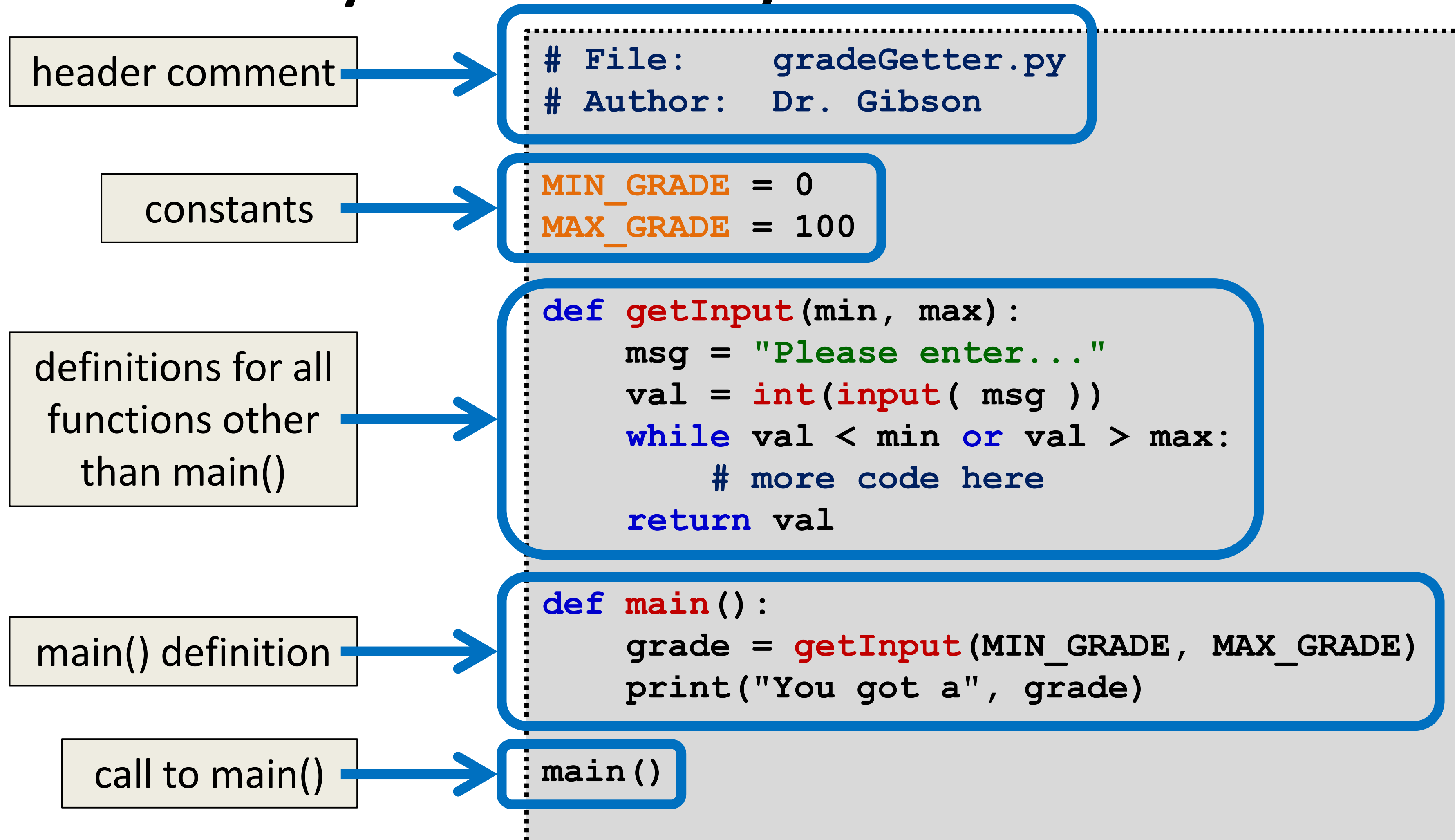


# Function Vocabulary



# File Layout and Constants

# Layout of a Python File



# Global Constants

- ***Globals*** are variables declared outside of any function (including **`main()`** )
- Accessible globally in your program
  - To all functions and code
- Your programs <u>may not</u> have global variables
- Your programs <u>may</u> use global **constants**
  - In fact, constants should generally be global

# Return Statements

# Giving Information to a Function

- Passing parameters provides a mechanism for initializing the variables in a function
- Parameters act as ***inputs*** to a function
- We can call a function many times and get different results by changing its parameters

# Getting Information from a Function

- We've already seen numerous examples of functions that return values

  **`int()`** , **`str()`**, **`input()`**, etc.

- For example, **`int()`**
  - Takes in any string as its parameter
  - Processes the digits in the string
  - And returns an integer value

# Functions that Return Values

- To have a function return a value after it is called, we need to use the **`return`** keyword

```
def square(num1):
    # return the square
    return (num1 * num1)
```

# Handling Return Values

- When Python encounters **`return`**, it
  - Exits the function (immediately!)
    - Even if it's not the end of the function
  - Returns control back to where the function was called from

- The value provided in the return statement is sent back to the caller as an ***expression result***

# Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

Step 1: Call **`main()`**

# Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

Step 1: Call **`main()`**

Step 2: Pass control to **`def main()`**

# Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

Step 1: Call **`main()`**
Step 2: Pass control to **`def main()`**
Step 3: Set **`x = 5`**

# Code Trace: Return from **`square()`**

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

Step 1: Call **`main()`**
Step 2: Pass control to **`def main()`**
Step 3: Set **`x = 5`**
Step 4: See the function call to **`square()`**

# Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():
    x = 5
→   y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

Step 1: Call **`main()`**
Step 2: Pass control to **`def main()`**
Step 3: Set **`x = 5`**
Step 4: See the function call to **`square()`**
Step 5: Pass control from **`main()`** to **`square()`**

# Code Trace: Return from **square()**

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

num1: 5

Step 1: Call **main()**
Step 2: Pass control to **def main()**
Step 3: Set **x = 5**
Step 4: See the function call to **square()**
Step 5: Pass control from **main()** to **square()**
Step 6: Set the value of **num1** in **square()** to **x**

# Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

num1: 5

Step 1: Call **main()**
Step 2: Pass control to **def main()**
Step 3: Set **x = 5**
Step 4: See the function call to **square()**
Step 5: Pass control from **main()** to **square()**
Step 6: Set the value of **num1** in **square()** to **x**
Step 7: Calculate **num1 * num1**

# Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
→   return num1 * num1
```

num1: 5

Step 1: Call **`main()`**
Step 2: Pass control to **`def main()`**
Step 3: Set **`x = 5`**
Step 4: See the function call to **`square()`**
Step 5: Pass control from **`main()`** to **`square()`**
Step 6: Set the value of **`num1`** in **`square()`** to **`x`**
Step 7: Calculate **`num1 * num1`**
Step 8: Return to **`main()`** and set **`y`** = return statement

# Code Trace: Return from **square()**

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

Step 1: Call **main()**
Step 2: Pass control to **def main()**
Step 3: Set **x = 5**
Step 4: See the function call to **square()**
Step 5: Pass control from **main()** to **square()**
Step 6: Set the value of **num1** in **square()** to **x**
Step 7: Calculate **num1 * num1**
Step 8: Return to **main()** and set **y** = return statement
Step 9: Print value of **y**

# Testing: Return from `square()`

```
>>> print(square(3))
9
>>> print(square(4))
16
>>> x = 5
>>> y = square(x)
>>> print(y)
25
>>> print(square(x) + square(3))
34
```

# Island Example

1. Function **`square()`** is called
   a. Make copy of **`x`**'s value (no name yet)
   b. Pass value of **`x`** to **`square()`**

5

X = 5

main()

USS Actual Params

1. Function **`square()`** is called
   a. Make copy of **`x`**'s value (no name yet)
   b. Pass value of **`x`** to **`square()`**

X = 5

5

main()

USS Actual Params

1. Function **`square()`** is called
    a. Make copy of **`x`**'s value (no name yet)
    b. Pass value of **`x`** to **`square()`**

1. Function **`square()`** is called
   a. Make copy of **`x`**'s value (no name yet)
   b. Pass value of **`x`** to **`square()`**
   c. Assign value to **`num1`**

1. Function **`square()`** is called
    a. Make copy of **`x`**'s value (no name yet)
    b. Pass value of **`x`** to **`square()`**
    c. Assign value to **`num1`**
    d. Execute **`num1 * num1`**
        a. (No name for it yet)

1. Function **`square()`** is called
   a. Make copy of **`x`**'s value (no name yet)
   b. Pass value of **`x`** to **`square()`**
   c. Assign value to **`num1`**
   d. Execute **`num1 * num1`**
      a. (No name for it yet)
   e. Return calculated value

25

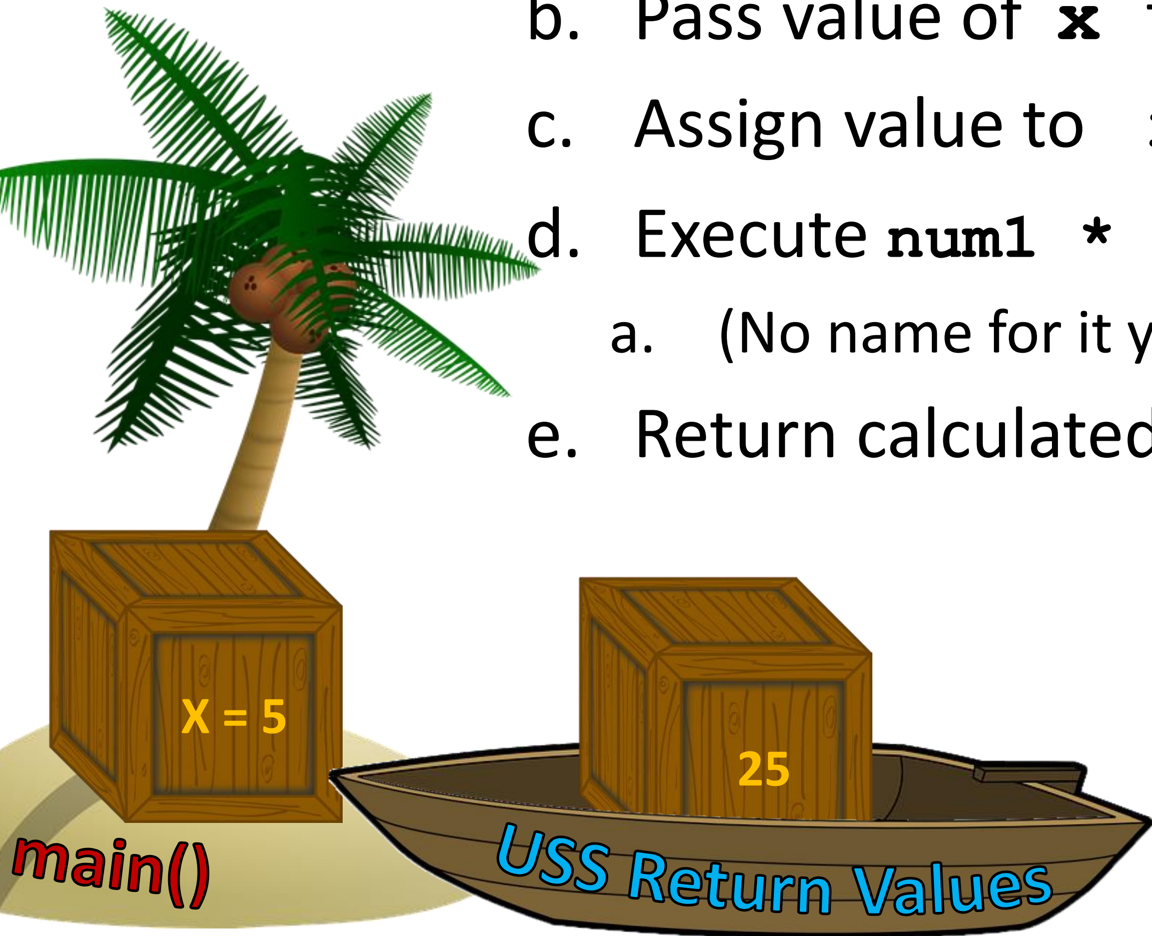
num1
= 5

25

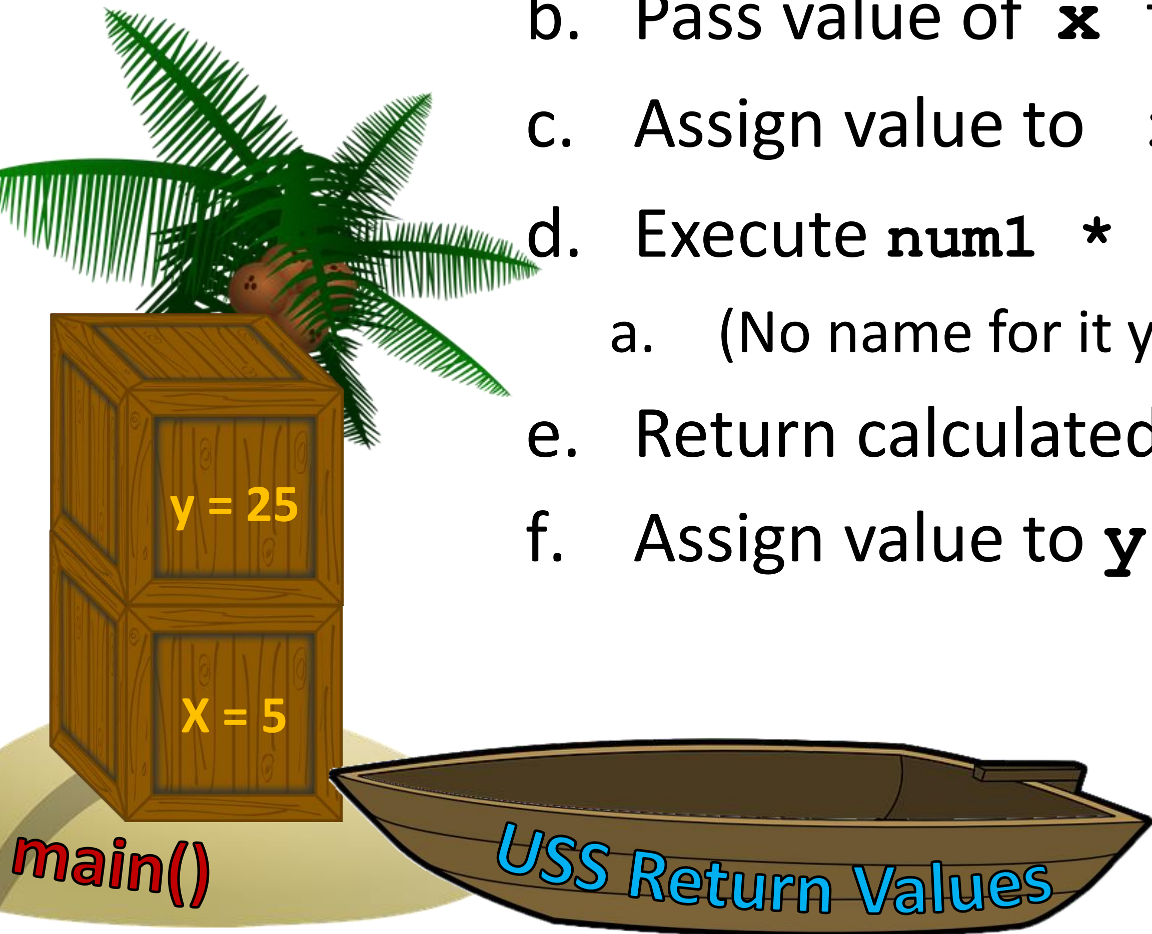
USS Return Values

square()

1. Function **`square()`** is called
   a. Make copy of **`x`**'s value (no name yet)
   b. Pass value of **`x`** to **`square()`**
   c. Assign value to **`num1`**
   d. Execute **`num1 * num1`**
      a. (No name for it yet)
   e. Return calculated value



1. Function **`square()`** is called
    a. Make copy of **`x`**'s value (no name yet)
    b. Pass value of **`x`** to **`square()`**
    c. Assign value to **`num1`**
    d. Execute **`num1 * num1`**
        a. (No name for it yet)
    e. Return calculated value
    f. Assign value to **`y`**





Images from pixabay.com

# `None` and Common Problems

# Every Function Returns *Something*

- All Python functions return a value
  - Even if they don't have a **`return`** statement

- Functions without an explicit **`return`** pass back a special object, called **`None`**
  - **`None`** is the absence of a value

# Common Errors and Problems

- Writing a function that returns a value but…

- Forgetting to include the **`return`** statement

```
>>> def multiply(num1, num2):
...     print("doing", num1, "*", num2)
...     answer = num1 * num2
>>> product = multiply(3, 5)
doing 3 * 5
>>> print(product)
None
```

Variable assigned to the return value will be **`None`**

# Common Errors and Problems

- Writing a function that returns a value but…
- Forgetting to assign that value to anything

```
>>> def multiply(num1, num2):
...     print("doing", num1, "*", num2)
...     return num1 * num2
>>> product = 0
>>> multiply(7, 8)
doing 7 * 8
>>> print(product)
0
```

The variable **`product`** was not updated; the code should have read **`product = multiply(7, 8)`**

# Common Errors and Problems

- If your value-returning functions produce strange messages, check to make sure you used the **`return`** correctly!

```
TypeError: 'int' object is not iterable

TypeError: 'NoneType' object is not
iterable
```

# “Modifying” Parameters

# Bank Interest Example

- Suppose you are writing a program that manages bank accounts
- One function we would need to create is one to accumulate interest on the account

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance
```

# Bank Interest Example

- We want to set the balance of the account to a new value that includes the interest amount

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance

def main():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
main()
```

What is the output of this code?

`1000`

Is this what we expected?

 Image from pixabay.com 

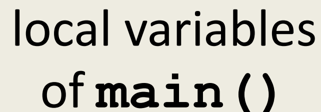
# What’s Going On?

- It was intended that the 5% would be added to the amount, returning $1050
- Was $1000 the expected output?

- No – so what went wrong?
- This is a very common mistake to make!
  - Let’s trace through the code and figure it out

# Tracing the Bank Interest Code

- First, we create two variables that are local to **`main()`**

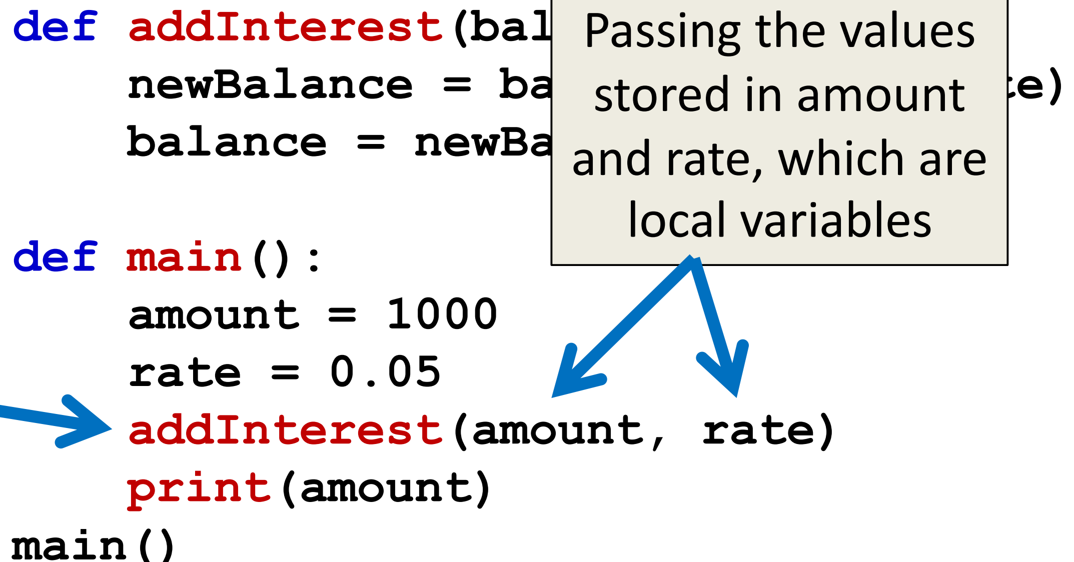



```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance

def main():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
main()
```

# Tracing the Bank Interest Code

- Second, we call **addInterest()** and pass the values of the local variables of **main()** as actual parameters

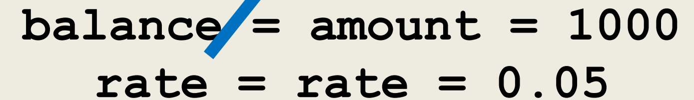


# Tracing the Bank Interest Code

- Third, when control is passed to **`addInterest()`**, the formal parameters of (balance and rate) are set to the actual parameters of (amount and rate)

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance

def main():
    amount = 1
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
main()
```



# Tracing the Bank Interest Code

- Even though the parameter **`rate`** appears in both **`main()`** and **`addInterest()`**, they are two <u>separate</u> variables because of scope

Even though rate exists in both **`main()`** and **`addInterest()`**, they are in two separate scopes

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance

def main():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
main()
```

# Scope

- In other words, the ***formal parameters*** of a function only receive the values of the ***actual parameters***
- The function does not have access to the original variable in **`main()`**

# Updating Bank Interest Example

# New Bank Interest Code

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    return newBalance

def main():
    amount = 1000
    rate = 0.05
    amount = addInterest(amount, rate)
    print(amount)
main()
```

# New Bank Interest Code

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    return newBalance

def main():
    amount = 1000
    rate = 0.05
    amount = addInterest(amount, rate)
    print(amount)
main()
```

These are the only parts we changed

# Announcements

- HW 4 is out on Blackboard now
  - All assignments will be available only on Blackboard until after the due date
  - Complete the Academic Integrity Quiz to see it
  - Due by Friday (March 3rd) at 8:59:59 PM

- Project 1 will come out this weekend
  - Read it closely, but do <u>not</u> start on it yet!
  - We will discuss the (required) design in class